Fast Vector Processing Language

Programming Languages and Translators
Department of Computer Science
Columbia University
Fall 2008

Gowri Kanugovi
Pratap Prabhu
Ravindra Babu
Topics Covered

- Language overview
- Tutorial
- Example program
- Architectural design
- Compiler implementation and features
- Testing
- Results
- Conclusion
Language Overview

- Computation of vectors like primitive types
- C like syntax and semantics
- Easy programming and fast execution
- Transparent and efficient utilization of SIMD instructions for computing at higher speeds
- SIMD instructions in x86 are called SSE and operates on 128 bits of data
- Data Parallelism is the goal of FVPL
Tutorial

- Basic data types, language constructs, types of operators and expressions, scopes are similar to the C programming language
- Consider the following program to add a vector to itself, assign the result to another vector and then display the result

```c
int main()
{
    int a[10], b[10];
    a = 5;
    b = a + a;
    print(b);
}
```
Tutorial (cont..)

- To execute the program, use the command
  
  $ ARGS="add_vector.fvpl" make exec; ./a.out

- The generated output would be:

```plaintext
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
```
Program demonstrating vector operations

```c
int main()
{
    int a[10], b[10], c[10];
    fread_int("aFile", a, 10);
    fread_int("bFile", b, 10);
    c = a&b;
    b = c+a;
    fwrite_int("cOut", c, 10);
    fwrite_int("bOut", b, 10);
}
```
Architecture
Architecture (cont..)

- **Lexer**: Scans the FVPL source code to generate tokens based on a set of rules.
- **Parser**: Parses input sequence of tokens from lexer to determine if the program is syntactically correct. Generates the AST.
- **Interpreter**: Walks the AST and checks if program is semantically correct. Populates the symbol table.
- **Code generation**: Translation of vector operations to function calls. Optimized functions reside in C++ stub code. Generated C++ code passed to the g++ compiler.
Compiler Implementation

- OCaml Lex
- Ocaml Yacc
- Eclipse (with OCaLIIDE)
- CVS (cvs.sourceforge.net)
- Shell script for testing
Compiler Features

- Generates C++ code from the source FVPL code
- Optimized routines to handle vector operations as part of stub code
- Stub code added as library to the generated C++ code
- Generated code compiled with g++
- Modular implementation
Testing

- Regression testing one at all stages using a shell script
- Positive and negative test cases written as part of the test suite
- Positive tests: Ensures generated C++ code and output is consistent for any valid input
- Negative tests: Ensures that the compiler catches all error conditions and provides the programmer with appropriate error messages
Testing (cont..)

- Shell script `testscript.sh` automates the testing process.
- Runs both the positive and negative test cases when invoked by the `make test` command.
- Excerpt of the generated output:

```
---------- Testing FVPL success cases ----------
tests/success/test-comments.input --------------- SUCCESS
tests/success/test-continue-break.input --------------- SUCCESS
tests/success/test-fibonacci.input --------------- SUCCESS
tests/success/test-file-input.input --------------- SUCCESS
tests/success/test-for.input --------------- SUCCESS
...

---------- Testing FVPL failure cases ----------
Fatal error: exception Failure("Cannot have more than one main function")
tests/failure/test-2mains.input --------------- FAIL
Fatal error: exception Failure("number of params different 2")
tests/failure/test-func1.input --------------- FAIL
Fatal error: exception Failure("Type of arguments not matching defined function3")
tests/failure/test-func2.input --------------- FAIL
...
```
## Source Code Statistics

<table>
<thead>
<tr>
<th>File</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanner.mll</td>
<td>76</td>
</tr>
<tr>
<td>parser.mly</td>
<td>156</td>
</tr>
<tr>
<td>ast.mli</td>
<td>76</td>
</tr>
<tr>
<td>interpret.ml</td>
<td>358</td>
</tr>
<tr>
<td>symbol.ml</td>
<td>140</td>
</tr>
<tr>
<td>fvpl.ml</td>
<td>10</td>
</tr>
<tr>
<td>stub.h</td>
<td>867</td>
</tr>
<tr>
<td>stub-print.h</td>
<td>131</td>
</tr>
<tr>
<td>Makefile</td>
<td>38</td>
</tr>
<tr>
<td>testscript.sh</td>
<td>39</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1852</strong></td>
</tr>
</tbody>
</table>
## Test Case Statistics

<table>
<thead>
<tr>
<th>Positive Test Cases</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of test cases</td>
<td>40</td>
</tr>
<tr>
<td>Number of lines in input files</td>
<td>505</td>
</tr>
<tr>
<td>Number of lines in output files</td>
<td>743</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Test Cases</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of test cases</td>
<td>20</td>
</tr>
<tr>
<td>Number of lines in input files</td>
<td>123</td>
</tr>
</tbody>
</table>
Results

- FVPL versus GCC compiler
  - Time Taken by C++ code to add brightness to 5 MB image = 416.994ms
  - Time Taken by FVPL to add brightness to 5 MB image = 212.091ms
  - Time Taken by C++ code to mask and multiply pixel (5 MB image) = 568.638ms
  - Time Taken by FVPL code to mask and multiply pixel (5 MB image) = 259.936ms

- FVPL shows 2X speed up in our tests
- Performance limited due to redundant data copy
Conclusion

- Using FVPL makes vector operations very simple
- Fast vector processing is important in Image Processing, Matrix operations, String matching and Cryptography
- FVPL does not exploit the full potential of SSE.
Conclusion (cont..)

- Useful enhancements to FVPL
  - 16 byte alignment
  - In-place and not in-place operations
  - Multi-dimensional array support
  - Operation on partial array
  - Intra-procedural data redundancy check

- FVPL leading the way for future architectures like Larrabee